USING MESSAGE QUEUING TO DRIVE POLYGLOT IDENTITY AND ACCESS MANAGEMENT DEVELOPMENT

A Paper
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By
Richard Ernest Orvin Frovarp

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

August 2016

Fargo, North Dakota

NORTH DAKOTA STATE UNIVERSITY

Graduate School

Title

USING MESSAGE QUEUING TO DRIVE POLYGLOT IDENTITY AND ACCESS MANAGEMENT DEVELOPMENT

By

Richard Ernest Orvin Frovarp

The supervisory committee certifies that this paper complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Brian Slator
Chair

Anne Denton

Bernhardt Saini-Eidukat

Approved:

July 31, 2017
Date

Kendall Nygard
Department Chair

ABSTRACT

This paper describes a method of using an Advanced Message Queuing Protocol (AMQP) broker, RabbitMQ in particular, to facilitate the management of accounts across a variety of systems. Higher education poses a unique challenge in the management of accounts due to the wide variety of systems involved. The Central IT department of an organization, those that usually run management systems, does not always have control over what systems are chosen, but needs to be able to manage them nonetheless. Unique requirements of each of the systems requires custom integration. That integration can be limited in what platform or languages are used.

Use of an AMQP broker along with JSON allows an identity management system to distribute changes in a platform independent and distributed manner. Administrators of systems are then free to choose their best platform and language for management.

ACKNOWLEDGEMENTS

I would like to thank all of those that helped make this possible. I would like to thank my advisor and friend Dr Brian Slator for his advice and guidance over these years, and his patience in completing this process. I would also like to thank Dr Anne Denton and Dr Bernhardt Saini-Eidukat for being members of my committee, and for their advice and encouragement.

My employers have made it possible for me to pursue this degree and projects that provide interesting challenges such as those presented in this paper. Thank you to Jody French and John Gieser of EduTech. Thank you to Britt McAlister, Dr James Ross, Marc Wallman, Galen Mayfield, and Steve Sobiech of NDSU's Enterprise Computing and Infrastructure. Thank you to Dale Summers, Jill Peterson, and Tim Mooney for their continued assistance, always patiently answering my questions, and helping to find and fix flaws in proposed solutions.

Finally, I would like to thank my parents Steve and Mary Frovarp, my brother and sister in law Michael and Andrea Frovarp, Emily Hagemeister, and all of my friends and family that have offered encouragement and support.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1. Motivation

Identity and Access Management (IAM) has become involved with more systems and services at NDSU. The custom engineered solution developed to suit the needs of locally hosted Unix and Linux systems was not a good fit for performing changes on remote or cloud hosted solutions. A new provisioning system was needed that could scale and would require less maintenance to keep running. Cost of commercial systems was a significant factor contributing to continued expansion of the existing solution.

University of Hawaii published a page [1] describing their use of an open source message broker, RabbitMQ [2], with IAM. Their use case is to broadcast changes in user information, allowing subscribers from different departments to be notified of changes to names and other information. This solution in this paper instead targets messages to only those systems that need to make a change for a specific service.

Finally, the original system at NDSU was imperative, dictating the changes that needed to be made. This can cause unnecessary errors when integrating remote systems due to communication errors. This solution has moved to an declarative methodology, which was inspired by Puppet [3].

1.2. Organization

This paper is divided up into several chapters. The second chapter provides a background of the problem domain, the history of the problem at NDSU, and the technologies used in the solution.

Chapter three provides a literature review of the current solutions, and the technologies used in the solution.

Chapter four is the approach and methodology of the solution. It describes how the various technologies described in chapter two are combined to solve the IAM problems outlined.

Chapter five provides examples of how the solution is used to solve real world problems, encountered by IAM engineers at NDSU. It also compares the proposed solution to those available on the market.

Finally, chapter six covers issues and future work.

1.3. Problem Statement

NDSU, like many other large organizations, developed a custom engineered Identity Management (IdM) solution in the 1990's. This solution was designed to drive account changes on Unix systems. It was developed using the appropriate tools of the time, including SSH, Perl, and shell scripts. While the solution worked well for Unix and Linux systems, it saw performance and platform issues with SSH against Java and/or Microsoft Windows. In addition, there was development overhead to add a new service due to the specific needs of the platform, making it difficult for administrators to use the proper programming language for the task.

The solution was to develop a system that made account changes on a variety of platforms, using a variety of programming languages. Given budgetary constraints, this solution needed to make use of open source components. The developed solution was to use RabbitMQ to distribute account change messages in the JSON format. This allows administrators to use a wide variety of languages, with Perl, PowerShell, Java, and Go currently in use at NDSU. The message format and contents were standardized, lowering the barrier to add a new service. The use of RabbitMQ enhanced resiliency and performance while allowing modifications to be chained. Complex services such as Microsoft Office 365 can be easily handled by chaining modifications.

Unlike commercial and open source solutions using the connector frameworks, it is easy to integrate languages that are not part of the .NET or Java ecosystems.

The work performed on the solution for this paper was done under employment of the Enterprise Computing and Infrastructure department at North Dakota State University.

2. BACKGROUND

2.1. IAM

Identity and Access Management (IAM) systems are used to control accounts, and the access that they have. Identity refers to the account, the user, and any number of authentication methods that a user or account may have. Not all accounts may map to a single user, as they may be shared accounts or service accounts that operate independent of any user interaction. With the exception of service accounts, shared accounts are considered a security risk. A single user may have several identities and authentication methods. Some of these identities may be hidden, or appear to the user to just be variations of the same thing. For example, a user may log into email via their username@example.com, but just username into another service. Security Assertion Markup Language (SAML) [4] backed services, such as InCommon [5], may expose a variety of identities, in part to control for (changeable) mutable usernames. Some of those identities may be unknown or even invisible to the user, each possibly employing different authentication methods. Examples of authentication methods that a user may have include the common password, an X.509 public key infrastructure certificate [6] for wireless, and multi factor.

IAM systems were previously known as IdMs, since the primary concern was with Identity Management. However, as systems become more complex, more widespread, and we develop a higher concern about security, the focus turns to include managing access in addition to identities. It is not sufficient to simply create an account in Microsoft Active Directory Domain Services (AD DS) [7] and do no more. Given the proliferation of Software as a Service (SaaS), more and more systems outside of the enterprise's physical systems need access to accounts, and to determine the level of access those accounts need. IAM plays a central role in managing accounts and enforcing security. The end points a user or system accesses are called services in the NDSU model.

In small organizations, IAM might be a person manually doing the work. In an organization of 50 people, it is possible to know when new employees are hired, when

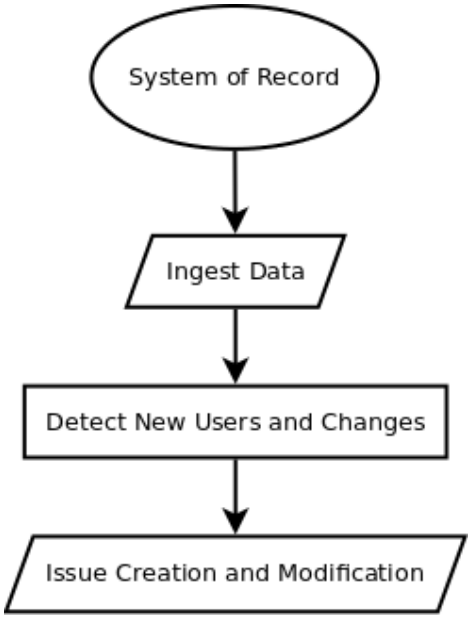

Figure 2.1. IAM Workflow

employees leave, and when roles change. As organizations grow, the need to automate more of the workflow grows.

The general workflow for an IAM system is:

1. Start with the system(s) of record. This is most likely human resources (HR) data, and in education would include student data.
2. Ingest data from the system(s) of record for who are officially affiliated with the organization.
3. Detect new users and provision services as appropriate according to the information in the data. Provisioning is the act of creating the accounts, roles, and permissions in systems.
4. Issue modifications for already provisioned users based off of changes in the data. A name change might update their information, and a new department or role might provision new services.

There is also the need to support accounts and provisioning workflow for those that are not paid by the organization (thus not in the HR database). Finally, every IAM plan should

include a workflow for deprovisioning the accounts. Deprovisioning is the act of removing access to a system. For some organizations this may be when HR data indicates that the person has left. In more challenging situations such as higher education, decisions to deprovision may be dependent on factors such as the semester calendar.

2.2. IAM at NDSU

2.2.1. History

NDSU has employed a custom engineered IdM/IAM system for over two decades. When the NDSU system was first developed, commercial alternatives were not readily available on the market, and open source solutions would not have existed. Since that time, there have been several commercial solutions on the market, all with a high price tag.

The first systems at NDSU were focused on creating accounts on the email system, which at the time were running Digital UNIX [8] and AIX [9]. As Unix systems, creation of the accounts could be performed by Secure Shell (SSH) [10], a network protocol allowing administrators to securely access remote systems. As time moved on over the following decade, most every system that needed management was a Unix or Linux [11] based system. This made SSH a perfectly fine solution.

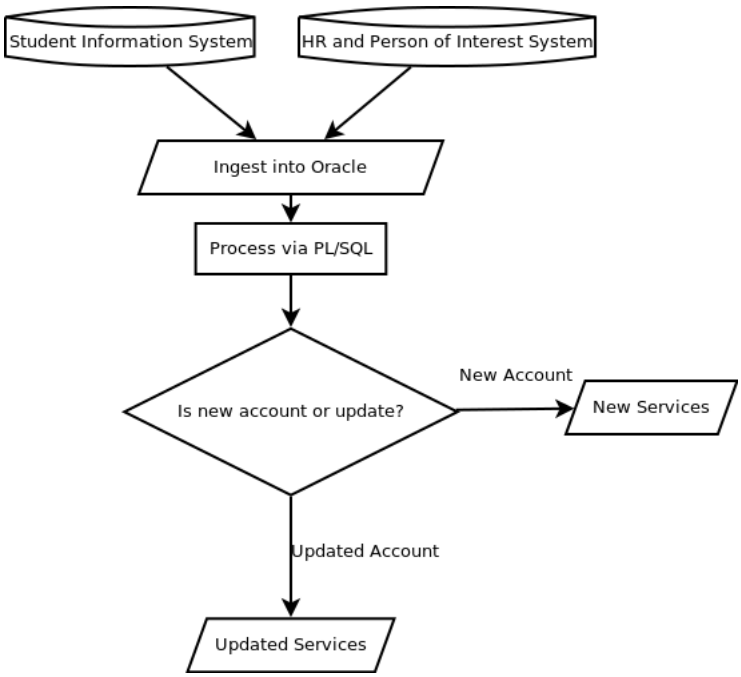

Figure 2.2. IAM Ingestion at NDSU

The operation of the system is as follows:

1. Feed files are downloaded once a day in the morning from the North Dakota University System (NDUS) transfer system. One file contains student information, and the other two files contain HR information.

2. These files are ingested using Oracle's tooling.

3. The resulting tables are processed using PL/SQL (Procedural Language/Structured Query Language) [12], an Oracle [13] extension to SQL [14].

4. These changes may result in new services being provisioned, or modifications to old services. Those changes are recorded to one of two different tables. One table for enrollments, the other table to indicate modifications. Collectively these tables, those in step three, and more are known as the User Services Database (USDB). See the Entity Relationship (ER) diagram Figure 4.1 on page 19.

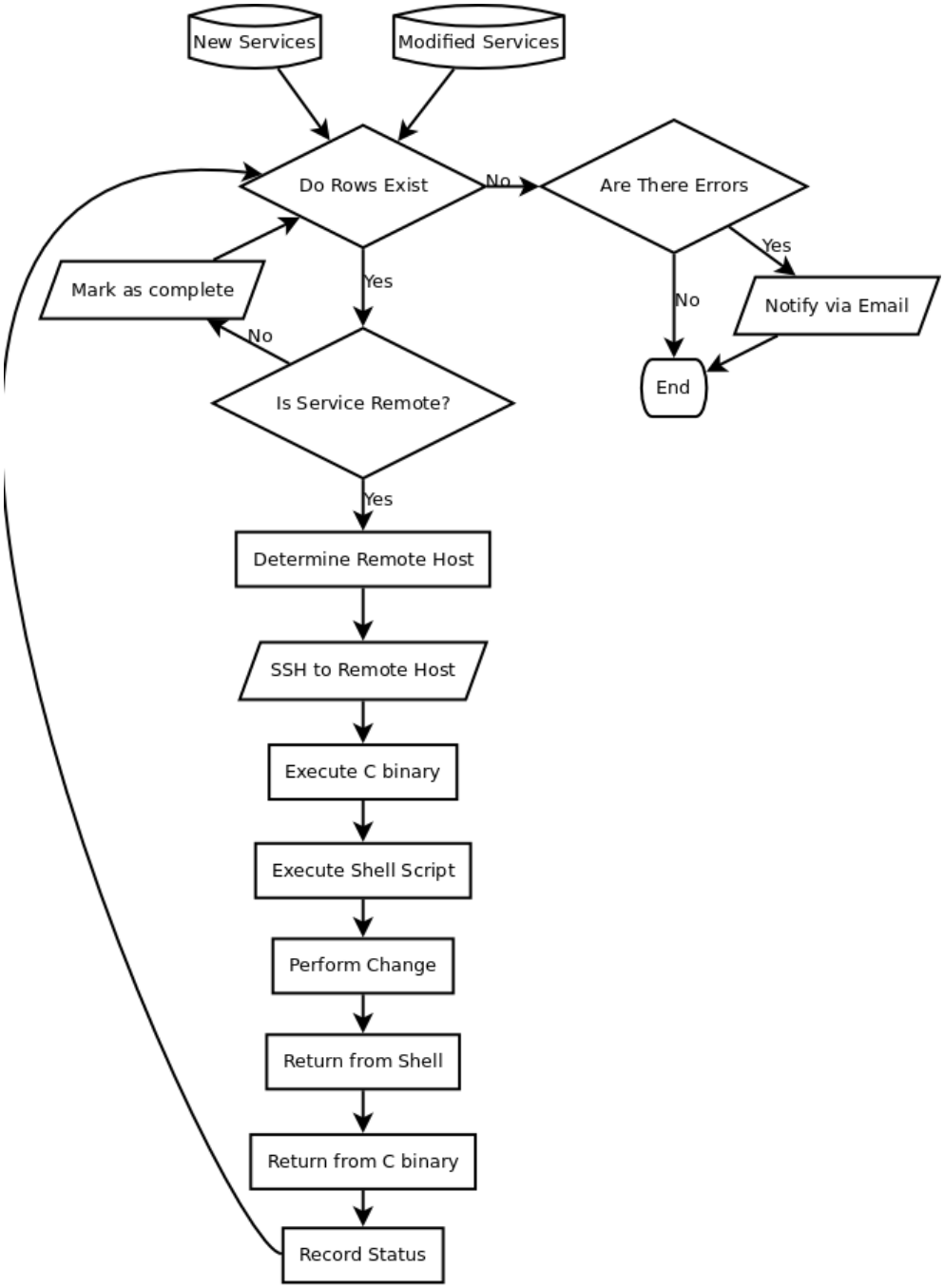

Figure 2.3. UAMS Operation

From there a separate process runs to operate on the tables modified in step four.

5. Perl [15] code running once every ten minutes checks those tables for new actions to perform. This is the dispatching User Account Management System (UAMS) code.

6. New services or modifications to services are compared against the services table to determine if the service requires remote modification. If it does not, the service modification is marked as complete, and next row is checked.

7. Those services that need remote modification are looked up against the remote host the service is running on.

8. SSH with an SSH key is used to execute a C binary to validate the input and protect against unauthorized remote code execution.

9. The C binary calls a shell [16] script that processes the specially formatted input.

10. The shell script, using configuration on the remote system, calls more shell code called a UAMS driver. The UAMS driver then performs whatever actions are necessary for that service. These actions may be to modify a database (DB), use Linux account commands, or run special Java code to control the Learning Management System (LMS).

11. The exit of the shell code passes control back to the C binary, which passes it back via SSH to the calling UAMS Perl code. On success exit code, the UAMS code updates the database and goes to step 6 to find new work. On failure exit code, the Perl code records a returned error string, and goes to step 6 to find new work.

12. After all rows have been processed, if there have been any errors, an email is sent to the administrators.

UAMS drivers perform the necessary changes on remote systems. In other IAM solutions, these would be called connectors [17].

There are other management steps outside of the feed files that can trigger changes. IT Security can choose to provision a specific service to a specific user at any point in the day. Account activations also trigger changes throughout the system.

The process from step six on runs sequentially. If one request takes a long time, each request behind it must wait. On the other hand, changes against Unix or Linux accounts were very quick to complete, and easy to write. In addition changes directly against a database were also easily possible by using the MySQL [18] command line tool.

This model starts to break down when the change is being made against the Blackboard LMS [19]. At the time, the UAMS driver for it would launch a specially crafted Java [20] application that effectively loaded a lot of the LMS into it for access to the Application Program Interface (API). The startup of the Java Virtual Machine (JVM) would consume a large amount of the time required to make any change to the system. After a single change was made, the synchronous return path would run, and the JVM would shut down. The next request would have to start the whole workflow over again, including spawning the JVM. During normal operations, this would not normally be visible to end users. However, during account purges when thousands of changes are being made, everything else would slow down noticeably. If a user tried to activate their account during that time, it might be hours before their activation changes were pushed through the system as everything else waited for Blackboard to go through all of its changes.

With the addition of Microsoft Windows Server [21] based systems, which are not a naturally good fit for what is effectively Remote Procedure Call (RPC) via SSH, the model became even more difficult to use. Complicating matters was that the first use against Windows via SSH was done against Microsoft Business Productivity Online Standard Suite (BPOS) [22] using PowerShell [23] remote cmdlets. The initialization and startup costs in terms of time for those cmdlets was even higher than the JVM load cost of Blackboard. Additionally, Microsoft throttled the number of new connections a system / service account could create in a unit of time.

What was needed was a language- and platform-agnostic solution that performed either in parallel or asynchronously.

2.2.2. IAM in Higher Education

IAM in higher education poses several unique challenges. The roles of a single user can vary and differ over time. Students become employees, and employees become students. Sometimes both roles are held at the same time.

The larger issue is the vast variety of systems that can need IAM controls. Commercial IAM solutions tend to focus on the most common and most prevalent types of services such as AD DS / Lightweight Directory Access Protocol (LDAP) [24], Exchange [25] /

Office 365 [26], Google Apps [27], and eDirectory [28]. Commercial organizations can direct the use of systems towards those that are friendly with AD DS.

Higher education runs a much larger variety of service types. While there are the normal services such as AD DS, Office 365, Google Apps, there are more specialized types of services. These may include LMS, survey systems, advising systems, activity tracking, and even bicycle checkout.

Central IT, where IAM systems typically reside due to their scope, scale, and complexity, often does not have much say over the systems that the university chooses to run. The ability to be flexible is of highest importance.

2.3. RabbitMQ

2.3.1. AMQP Broker

RabbitMQ is an open source AMQP 0-9-1 compatible message broker [29]. It is currently owned by Pivotal, and is offered under the Mozilla Public License version 1.1 [30]. It runs in the Erlang [31] Virtual Machine (VM).

AMQP brokers operate by a publisher publishing a message to an exchange, which then routes the message to the appropriate queues. Consumers can then read those messages out of those queues in a First In First Out (FIFO) manner.

2.3.2. Messages

A message in AMQP consists of two parts: headers and a body.

Headers contain any information that may be helpful to the processing of the message. Several uses of headers will appear in the next sections. The body is treated as an opaque block of binary data.

2.3.3. Routing

Messages are published to exchanges on virtual hosts (vhosts). Vhosts implement a method to segregate data on a single system. They exist to provide some level of partition, and to make management of the system easier.

Exchanges exist in virtual hosts. They are what perform the message routing. The exchange needs to be created for a message to be published to it. If no configurations match the routing key of the message, the message is silently discarded. If multiple configurations match, the message will be replicated into each queue. There are three types of exchanges:

- Fanout exchange - a message is sent directly to one or more configured queues.
- Direct exchange - a message is sent to zero or more configured queues based on a complete routing key match.
- Topic exchange - a message is sent to zero or more configured queues based on portions of the routing key.

A routing key is a special attribute that is stored in the header. In the direct exchange type, the routing key can be a simple arbitrary marker. Topic exchanges change the routing key into a period delimited set of words, that allows for the inclusion of two types of wild cards.

Routing keys for topic exchanges tend to look like dotted hierarchies that describe the contents in the message. The routing key for a message about Fargo's weather might be `northamerica.us.nd.cass.fargo.weather`. The special wildcard characters of `*` and `#` can be used to control routing of those messages to queues. `*` can substitute exactly one word, where `#` can substitute zero or more words. The `*` implementation is not aligned with the match zero or more usage of that symbol in most other implementations including SQL and regular expressions. `northamerica.us.#` would route all messages that are for the US. `northamerica.us.*.cass.#` would route all messages in the US that are for counties that have the name of Cass, regardless of state or city.

2.3.4. Queues

Queues contain the messages for the consumers to read. The messages in each queue are independent of any other queue. If a message is routed to five queues due to routing rules, five independent copies of those messages would exist in the system.

Multiple clients can connect as consumers to a single queue to read messages. The broker will ensure that each message in the queue will only be sent to one client. This allows multiple clients to connect to a queue to share the workload. If multiple clients need their own complete copy of the stream, multiple queues need to be used with appropriate routing.

The messages in the queues are handled in a FIFO manner. The specification has provisions for priorities overriding FIFO, but that is beyond the scope of this paper.

AMQP has several operations surrounding message retrieval. `basic.Get` is used to get one message at a time. `basic.Consume` is used to consume several messages at a time. When using `basic.Consume`, it is probably important to set `basic.QoS` to prevent all of the messages from being sent to one consumer in giant batches, potentially starving other consumers in a load balanced environment.

Consumers also have the choice of using auto-acknowledge (ACK) or not. Consumers using auto-ack automatically acknowledge the message upon reading the message. Acknowledging a message removes it from the queue. Consumers that do not auto-ack must explicitly acknowledge each message before it is removed from the queue. If the consumer is disconnected from the broker before the message is ACK'd, it is returned to the top of the queue. A consumer with auto-ack turned off has the ability to negatively acknowledge (NACK) a message, and push it back to the top of the queue.

2.3.5. Persistence

By default, much of the configuration, queues, exchanges, and messages themselves are not persistent in RabbitMQ. A restart of the broker means that everything disappears. In some cases this may be beneficial, but in the case where messages represent work that needs to be performed, it is desirable that the configuration and messages survive any outage that may happen.

Exchanges and queues can be declared as durable. Durable exchanges and queues survive broker restarts and persist until deleted.

Messages published to a durable queue will not necessarily survive a broker restart. For a message to survive a broker restart, the message itself must be declared as persistent. This declaration is made in the message headers. Such messages should be synced to disk before a graceful shutdown of the broker. An ungraceful shutdown may cause the message to be lost. Publishers can ensure that a message is synced to disk by either using transactions or setting the channel to ConfirmSelect. Both methods will slow the publishing process down, but are the best way to ensure that a message will survive an ungraceful shutdown, assuming the underlying disk survives. Of the two, ConfirmSelect is faster [32].

2.3.6. Connections

Connections to RabbitMQ are made through a variety of libraries in the developer's language of choice. Connections used by the solution in this paper include the official .NET and Java libraries, github.com/streadway/amqp [33] for Go, and AnyEvent::RabbitMQ [34] for Perl with University of Hawaii's wrapper [35]. Either official or unofficial libraries exist for most modern development languages is use. The .NET library can be used with PowerShell with some minor difficulty, see Chapter 6 for details.

Clients can connect via Transport Layer Security (TLS) [36]. The option to connect via plain text is available, but at this point in time, conforming to modern practices, everything should be encrypted.

Connection time for any client is quick. Code can be developed that periodically connects, checks for work in a queue, and disconnects if there are no messages in the queue. It is also simple to use the libraries to connect and block on empty queues and process messages as they arrive. By blocking on empty queues, the read message call will not return until a message is read. This is used to operate persistent connections to the broker, as AMQP is a stateful protocol.

Reconnection code varies depending on the language in use. If exchanges and queues are preconfigured, it is much easier to reconnect after a remote shutdown. streadway's Go library has such functionality largely built in, and Lyra [37] is with Java.

2.3.7. RPC

While not in use in this solution, it is possible to perform RPC via AMQP. The producer creates a callback queue, and sends information about that queue in the headers of the message. The consumer, which is still operating under FIFO, then sends the response back via the callback queue. This queue is most likely transient, and may be deleted after the one request is processed. There are provisions for one callback queue per producer, with correlation IDs being used to correlate request with response.

2.3.8. Configuration

Configuration of RabbitMQ can be done by the various producers and consumers as they connect, depending on permissions granted to the connecting usernames. This allows

for a very dynamic system in which the configuration can easily and rapidly change to meet needs. Consumers can join message streams of current activity with no history, for instance.

However, dynamic configuration has a drawback. In order for a published message to be delivered to a queue, a consumer has to connect, create the queue, and setup the binding of the exchange to the queue. This needs to be done before the producer publishes to the exchange, otherwise the consumer will never see the message. A solution to static configuration is presented in Section 2.5.

2.4. JSON

JavaScript Object Notation (JSON) [38] was first used for serialization of JavaScript objects. Strictly speaking it is a subset of YAML Ain't Markup Language (YAML) [39]. JSON has become one of the most popular methods of exchanging information between systems. Being object based, it can more easily handle more sophisticated data that flat file options such as Comma Separated Value (CSV) [40]. It is less verbose than eXtensible Markup Language (XML) [41]. While it can represent everything that XML can, it presents fewer options than XML, which makes parsing easier.

```
<person type="faculty">
  <firstName>User</firstName>
  <lastName>Name</lastName>
  <office phoneNumber="555-5555">
    <location>123 Any Hall</location>
  </office>
</person>
```

```
{
  "type": "faculty",
  "firstName": "User",
  "lastName": "Name",
  "office": {
    "phone": "555-5555",
    "location": "123 Any Hall"
  }
}
```

The use of attributes in XML allows for ambiguous design [42], which makes parser development and usage more complicated than JSON. Illustrated above in the XML, phone

number is an attribute of the office tag, but location is its own tag and the information is not in an attribute. Both of these are valid, and up to the designers of that particular XML document. Both allow for nested data, such as office shown above.

PowerShell 3+ has a built in JSON parser, ConvertTo-Json and ConvertFrom-Json [43], which operate on .NET dictionaries, which are simple key value stores, where the value can be another dictionary. Go also has a built in JSON parser, `encoding/json` [44], where values are stored in structs. Java does not have a built in parser, but there are two excellent Apache License 2 [45] licensed libraries for use: Jackson [46] and Google GSON [47]. Both use fully realized Java objects to serialize and deserialize. In Perl `JSON` [48] is used, and the resulting value is a Perl hash of hashes. The absence of attributes, makes storage to a dictionary / hash / associative array (PHP Hypertext Processor (PHP) [49]) simple to implement for the library and language developers, and easy to use for the developers.

2.5. Configuration Management - Puppet

Given that pre-existing deterministic configuration is needed before the first message is sent, a configuration management solution is a wise choice. This implementation uses Puppet to manage RabbitMQ, which is running on a VM using RedHat Enterprise Linux 6 [50].

The Puppet Labs RabbitMQ [51] module is used to manage the system. This module exposes several Puppet types, with the driving code developed in Ruby [52], the primary language of Puppet. Since RabbitMQ does not have a native command line utility to make all of the desired configuration changes, such as creating bindings, the module uses an additional command line utility to drive the optional web management interface.

To make management of the configuration easier, custom Puppet types were created that perform the default configuration necessary to fully wire in a service. This includes creation of the driver’s user, the durable queue and exchange, and bindings to get messages and send responses with the proper topics already configured.

3. LITERATURE REVIEW

3.1. Commercial and Open Source IAMs

There are several commercial and open source IAM solutions that either have been or currently are available. One of the early full feature systems was Sun's Identity Manager. This system was able to ingest data from multiple sources, trigger changes based on rules, and perform changes. Later, connectors were added to help perform changes on remote systems. Sun Identity Manager has been end of lifed, and replaced with Oracle products after Oracle's acquisition of Sun Microsystems.

Microsoft's Identity Manager [53], formerly Forefront Identity Manager (FIM), entered the market in 2010. Being from Microsoft, it is developed in .NET, and designed to interact with Windows based systems. Microsoft has released free versions of their systems to synchronize a local AD DS and to their Azure AD (AAD) [54]. One large limitation of FIM, that appears to be resolved with Identity Manager, is missing self service in FIM. To change a password, users would use their domained computers, rather than a self service website. This does not work in a Bring Your Own Device (BYOD) environment like higher education.

Several other solutions exist on the market in various forms of open source. Evolveum has performed a comparison of some of the options available on the market [55]. It should be noted that Evolveum is the developer of midPoint [56], an open source IAM solution. There may be some bias in their reporting. However, their assessment of licensing is fair and valid. The two realistic options in open source are midPoint and Apache Syncope [57]. Both solutions are licensed under the permission Apache License 2. It should be noted that the author is a member of the Apache Software Foundation [58], and is biased towards permissive licenses.

As is common in some open source projects, ForgeRock's OpenIDM [59] is available under the CDDL [60] license. Those using the CDDL option are not given the ability to use stable releases. For stable releases and support, a commercial license is required. The CDDL license would not allow a third party to offer support based on their product.

3.2. RabbitMQ

The RabbitMQ homepage proviedes a large amount of information on the project. Examples of how to perform various tasks are presented in a variety of languages. Currently the tutorials are available in nine languages, varying from the mainstream Java/C#/Ruby to the new Go to the more obscure of Elixir [61]. While the project documentation is good, the *RabbitMQ in Action: Distributed Messaging for Everyone* from Manning Publications [62] covers all of the basic topics and includes advanced topics such as monitoring with Nagios [63].

The AMQP spec is also available for reference.

3.3. Hawaii IAM

University of Hawaii operates their UH Message Broker service. This service uses RabbitMQ as its method of communication. Their usage helped inspire the solution presented in this paper. The goal of their system is to notify other systems of additions and changes of members and affiliates of their institution. Their solution allows services to connect and listen to messages that are of interest to the service. The UH Message Broker service transmits their messages in XML.

In addition to providing their service, University of Hawaii also provides a wrapper for Perl's `AnyEvent::RabbitMQ` package. This makes using RabbitMQ in Perl much easier.

3.4. Puppet

Puppet is a declarative configuration management system. Its declarative style provided the other half of the inspiration for the solution. This differs from the imperative style in use by the original UAMS solution.

In addition to providing inspiration, it is also used to manage RabbitMQ. Puppet can be used to manage any Linux based system, and also offers options for managing Windows Server.

The Puppet website provides a good overview of how puppet works. For development of custom types and providers, *Puppet Types and Providers* from O'Reilly [64] offers a good introduction.

4. APPROACH

4.1. Service Ordering

As more services were added to the system, it became important to handle service ordering. The need is easily demonstrated for the provisioning use case. A user cannot be added to an AD DS group until the user has an account in the system. The driver for the group would not know what the appropriate action is: should it wait for an account to show up, or emit an error indicating that the account does not exist? The provisioning options at NDSU, automatic and via web management systems, allow for several services to be requested at the same time. This means that the group and AD DS can be requested at the same time to make the request process easier, making it the responsibility of the provisioning system to sort out the order.

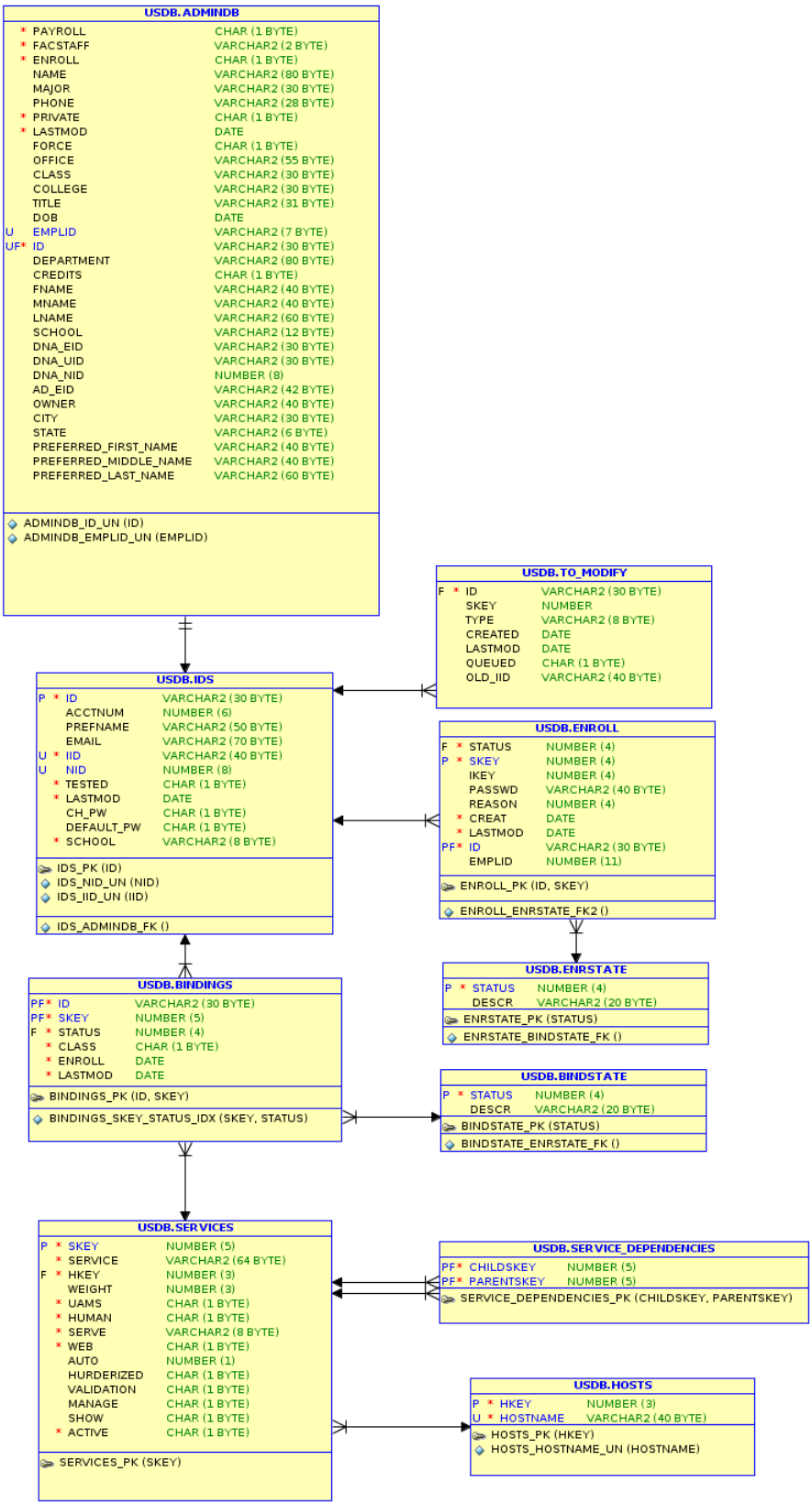

Figure 4.1. UAMS Entity Relationship Diagram

The `service_dependencies` table was added to the existing schema to indicate the prerequisites. Due to the requirements specified, if multiple prerequisites are listed and any one of them is met, it is then safe to provision. At the time, different services existed for email to differentiate between student and employee accounts, so the requirement of having an email account had to be satisfied by listing both in an either or manner. Initially the two email systems, BPOS for employees and Microsoft Live@edu for students, were so different, that two different Windows Servers running two different versions of the OS was required.

Initial deployment of the system only honored prerequisites for the provisioning action. Modification and deprovisioning did not honor the prerequisite options. Under most circumstances this has no undesirable effects. Determining ordering for modifications can be difficult due to the large number of reasons that modifications can be triggered. If the deprovisioning code considers the service being gone a success, ordering generally matters little. If the AD DS account is removed before the group is, due to how AD DS works that group membership would also be gone. So if the group driver gets the `remove` request after the account is removed, it will not find an account to remove the group from, and can return success.

However, when that group contains information to properly and completely deprovision the service, it is important that it is not removed prematurely. The implementation of user directories, by Gary Whaley, in Windows File Services randomly assigns a storage volume to the user to contain their data. To access their data, they are added to a group for that storage volume. Therefore, the group for that storage volume also indicates exactly which storage volume contains their data. The driver for this service uses that group to find and delete the data, freeing up space on the volume. If the group is removed early due to the account being deleted, removing data would be a lot more difficult. Due to this, service deprovisioning happens in the reverse order, inverting the prerequisites table. This addition has also acted as a safety mechanism to prevent accounts from being deleted unexpectedly when other services would have been affected in an undesirable way.

4.2. Operations

The operations of the system, UAMS2, match the operations of the previous system. This meant that no changes were required in most of the management code.

4.2.1. Create

The first operation performed on an account is the `create` operation. These operations are denoted by entries in the enroll table having a status of 2. The UAMS2 dispatcher monitors the table for changes. Before a `create` message is sent, the service is checked to see if it has any prerequisites, if not the message is sent. If the service has prerequisites, the account is checked to see if the prerequisite is met. If it is met, the request is sent, if it is not met, the service delayed until the next time when the process repeats itself.

4.2.2. Modification

The previous system was quite limited on what would trigger a modification. A service would only receive a modification request if that service was specifically impacted by a change. For instance, if a service did not need the email address, it would never receive a modification request when the email address changed. This in part was due to the fact that less than 10 services were controlled by this system at any time.

This solution triggers modifications on a large number of changes about a user. Names, phone numbers, titles, majors, email addresses, and more all trigger modifications to any services managed by the system. While an AD DS group may not need to be notified about the change of a first name on an account, AD DS itself along with Google Apps, Qualtrics [65], BikeShare [66], and others require notification.

The other important reason for triggering a modification request is to enable or disable the service for an account.

The `to_modify` table indicating that modification is required only denotes the service, the user, and a timestamp. That table would need to be expanded to include more information, then rules added to determine if a modification required that a driver would need to be notified. To make the system as simple as possible and to make it as easy as possible to add a service, all modifications are sent to all services managed by the system. If a service such as an AD DS group does not care about the `givenName`, it can simply check the enabled/disabled status and return success off of that alone.

4.2.3. Delete

Deprovisioning is performed to remove the account from the system. This removes all potential security issues with an account remaining, and can help free up space and licenses.

Delete messages are triggered by a service binding having its status set to 5, which correlates to `remove`. Messages are sent out in the inverse order for prerequisites to ensure that all resources are cleaned up appropriately.

4.3. Message Dispatch

The original system dispatched custom messages to each of the services being managed. Services were only sent the information that they needed to know. This works well for small numbers of services, but does not scale quickly and imposes a barrier to adding new services.

It was determined that most of the services would be well served by sending most of the information in a single message. The messages are in JSON and are largely identical across all services and operations, with only the service and actions changing.

```
{
   "legalGiven" : "Richard",
   "cn" : "Richard Frovarp",
   "nid" : "71866",
   "actionType" : "modify",
   "timestamp" : "2016-07-05T18:59:22Z",
   "legalSurname" : "Frovarp",
   "mail" : "richard.e.frovarp@ndsu.edu",
   "emplid" : "-510436",
   "iid" : "richard.e.frovarp",
   "commandType" : "enrollment",
   "skey" : "228",
   "locked" : true,
   "facstaff" : "N",
   "uid" : "frovarp",
   "college" : null,
   "oldIid" : null,
   "service" : "adauth",
   "suspended" : false,
   "sn" : "Frovarp",
   "class" : null,
   "givenName" : "Richard",
   "schoolServed" : "ndsu",
   "major" : null,
```

```
   "enroll" : "N"
}
```

The message above is to perform a `modify` on the `adauth` service, which represents the AD DS account at NDSU.

For those services that require a customized message, it is possible through additional development. The UAMS2 dispatcher is developed in Perl, and uses a dispatch table to determine which message generating subroutine to use. If a match is not found for the service in the dispatch table, the default message subroutine is used. All current custom subroutines also use the default subroutine as the base for their messages.

Examples of services using custom messages are:

- Google Apps is sent a default password upon provisioning
- Windows home directory is sent a quota
- Digital Measures [67] is sent the date of birth
- Office 365 is sent additional deliverable addresses
- North Dakota State College of Science (NDSCS) AD DS is sent a student or not determination to aid with Organizational Unit (OU) placement

4.4. Message Routing

4.4.1. Routing Keys

All message routing is done via the topic routing type. The routing key is determined using one of two different methods.

Service names are short human readable names to denote a service. They correspond to the `^[A-Za-z0-9_-]{1,32}$` regular expression.

All services developed under UAMS, and most still being created under UAMS2, follow the `^[A-Za-z0-9_]{3,32}$` regular expression. The same except lacking hyphens. The routing key pattern is: `<service>.<serviceKey>.enrollment.<operation>`.

`<service>` refers to the human readable service name described above. `<serviceKey>` is a numeric value that corresponds to a particular service. In some cases,

the same service name is applied to different instances of the service. See the beginning of section 5.1 for an example. `<operation>` corresponds to the operations defined in section 4.2.

Services with hyphens are treated with the `^([A-Za-z0-9_]+)(-[A-Za-z0-9_-]+)$` regular expression and grouping. The first group, before the first hyphen, is the `<prefix>`. The second group, which may contain more hyphens, is the `<remainder>` of the service. The routing key in this case is:
`<prefix>.<remainder>.<serviceKey>.enrollment.<operation>`.

These types of service names are used to denote wildcard services. This allows one driver to easily handle multiple services without additional configuration. If every AD DS group service name is of the form `ADS-<Group_Name>`, everything that starts with ADS can be sent to automatically sent to the same driver. If a particular AD DS group needs to be handled differently, it is created such that it does not match the `ADS-<Group_Name>` pattern, and is handled by a different driver.

4.4.2. RabbitMQ Routing Bindings

Bindings for the routing keys can be made to match the desired use case for that particular service. Since bindings are handled by Puppet and can be added to a live system, this removes the outage barrier present in UAMS.

Since most services are simple in configuration, the typical binding for the service is:
`<service>.#`

This routes all requests for the service, by name, to a single queue. This works for most systems. For those that are of the wildcard type, it would change to `<prefix>.#`

It is possible that multiple services can route to the same queue, and this is handled by adding more bindings. Some services may require that requests be sent to multiple drivers, and that is handled by adding more bindings to different queues.

More complicated systems can break the different actions into different queues. Office 365 `creates` can be slow, so it may be desired to have those actions split into a different queue. For Blackboard the `removes` can be slow, so those may be split off into a different queue. Since the action is included in the routing key, this is easy to do on a per service basis.

4.4.3. Configuration Management

Configuration of the users, permissions, queues, exchanges, and bindings for RabbitMQ is handled by Puppet, although any configuration management system should work.

The purpose of using configuration management is to ensure that the queues and bindings are setup before messages are sent for a service. Since each message is for work that needs to be done, each message needs to be stored and acted upon. If it is left up to the drivers to configure RabbitMQ themselves, it is possible that messages would be missed by the driver not being installed and started on time. In an IAM system, this has the potential to create security issues.

Since almost all drivers have identical RabbitMQ configuration, with the exception of username, password, and routing key, a Puppet type was created to make configuration more consistent and easier. This type:

- creates the user
- grants the user access to exchanges and queues named like the user
- creates an exchange to match the username
- creates a bindings to match the username
- creates a binding from the controller's exchange to the queue using the specified routing key
- creates a binding from the driver's exchange to the controller's queue using the `uams.*` routing key

If the driver handles multiple services, that are not wildcard type, the Puppet configuration uses the custom type, and then is configured to add additional bindings.

4.5. Failure Handling

Given the nature of the messages, failures need to be handled in a way that minimizes the chances of requests not being processed.

4.5.1. Dispatcher Handling

From the UAMS2 dispatcher there are two times when failures need to be handled: when the message is sent, and when the response is returned.

The entire dispatch mechanism is built around the possibility that systems can fail, and uses the best methods possible to handle those failures.

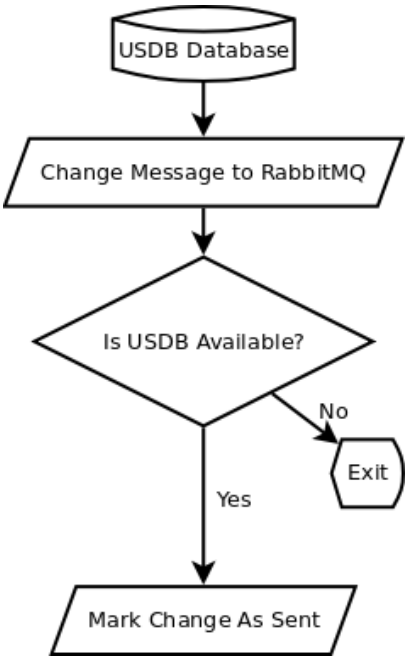

Figure 4.2. UAMS2 Message Dispatch

To dispatch a message:

1. Request is read out of the database.
    - While the tables differ between `create`, `remove`, and `modify`, the methodology is the same.
2. Message is sent to RabbitMQ with the header used to ensure that the message is persistent and with a method to ensure it is synced to disk.
3. The database is updated to indicate that the message is queued.
4. If the database is not available in step three, the process exits. Otherwise it continues to read requests out of the database until there are no more requests.

If the database goes down between steps one and three for a single request, this results in a request being sent, but not marked in the database. When the database becomes available again, the request will appear in the tables as though it was never sent. This will result in a duplicate message being sent. It is up to the driver to handle duplicate messages. Methodologies for handling duplicate messages are covered in the driver section.

There are two possible failure modes to handle when the response is returned. It is possible to receive duplicate responses for a single message being sent. If the dispatcher duplicates a request, it will have only one row for two responses. In this case, the dispatcher simply moves on if it does not have a matching row to update, by assuming that it is a duplicate response.

The other failure scenario is when the driver runs into a fatal error on the request and returns an error back. In this case, the error is logged, and the administrators are notified so that they can diagnose and handle the error.

For any failure scenario, if the request needs to be resent, the database is manually updated to remove the queued marker, and the message will be sent the next time the tables are queried.

4.5.2. Driver Handling

Due to a wide variety of potential communication errors, the drivers have a specific order of operations to handle requests.

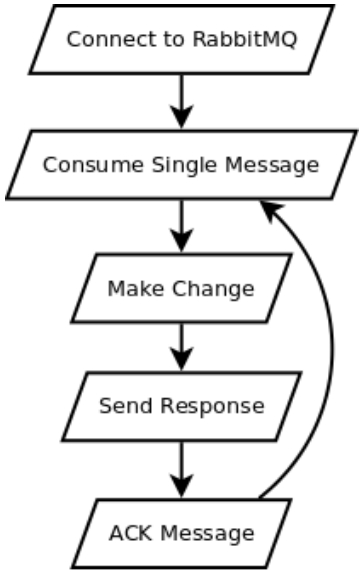

Figure 4.3. UAMS2 Driver Message Handling

1. Connect to RabbitMQ with auto-ack turned off.
2. Get or Consume a single message.
3. Perform the necessary action.
4. Send the success / failure message using a mechanism that ensures that the message is persisted and synced to disk.
5. ACK the message retrieved in step 2.

Optionally, the driver can connect to the remote resource and test its availability. Since there generally is not a message waiting, it is likely not efficient to perform this in advance of processing the first message.

If RabbitMQ is not available in step one, it simply tries again later. If the service is not up in step three, the message can be NACK'd or simply disconnect and the message will be there to try again later. If RabbitMQ is not available in step four, the driver will receive the same request again, as this means that RabbitMQ has disconnected, which will automatically NACK all outstanding messages. If RabbitMQ is not available in step five, once again the driver will receive the same request again as its current one has been automatically NACK'd.

This leads to the scenario above where the dispatcher will receive multiple success / failure messages for a single request.

Several of the scenarios described in this section and the previous result in duplicated request processing for the driver. This leads to a change in strategy as to how the driver should behave. With imperative UAMS, an error would be triggered if an account already existed for a create, or the account was already removed for a delete. The possibility of duplication of messages makes this strategy invalid.

Puppet is designed such that the configuration describes the desired end state of the system after Puppet runs. Describing a resource with `ensure => 'absent'` will typically delete the resource. However, Puppet will not throw an error if that resource has already been removed by a previous run.

This strategy has been integrated into the system. A `create` request means that the service will most likely have to be created for the user, but a driver should not throw an error if it already exists. It allows the driver to assume that the account does not exist, and proceed with creation instead of performing a search first. If the `create` fails because the account already exists, the driver should most likely call its modification code to ensure that the account is in the proper configuration. Some drivers with multiple steps may need to ensure that all of the steps have been performed.

For instance, if the `create` and license steps are two different web service calls, the driver should ensure that both have been completed. For particularly complex services, ordering of individual steps may be important to make it easier to pick up where the process last failed.

A driver that receives a delete should assume that the account is present, and do the necessary work to clean up. Once again, complex services may need to pay particular attention to the order of the clean up operations to ensure that all steps are performed. A delete on an already completely cleaned up account should return success as the desired state was that the account did not exist.

The one exception to these assumptions is the `modify` operation. A `modify` operation assumes that the account exists. A `modify` should not be sent on an account that has yet to be created, or an account that has been deleted. A driver that receives a `modify`

on an account it cannot find should return an error. The UAMS2 dispatcher has protections built in to prevent such a message from being sent. However, this does not prevent administrators of systems from manually removing accounts from systems without updating the IAM system. While this should not be done, the code should help make it obvious when administrators do not follow the rules.

The other failure scenario for a driver is when its resources are not available. For example, if a driver needs to connect to a remote web service, and that service is not available. The driver should not return errors for requests on that service. The errors need to be handled by the driver, which usually includes waiting for the resource to become available, or for the request quota to reset. The driver may choose to email its developers to flag an issue to be addressed. It is likely that not all service related issues will be caught in the development of the driver, and those may report errors. Handling of those errors should be added to the driver code to reduce the amount of noise created by those errors.

5. EVALUATION

5.1. Office 365 Use Case

Email has shaped the IAM solution at NDSU for approximately two decades. The email service existed under multiple service keys. This was to denote that a user had email by the service name, and which server their email account was on by the service key.

With the move to BPOS, the previous Unix / Linux based drivers no longer worked. BPOS was added as a UAMS driver with great pain. BPOS management required the use of PowerShell, which is not an ideal fit for an SSH based system. It was also very slow. This helped lead to the decision and inspiration to develop a system that was asynchronous and worked across languages and platforms.

5.1.1. Order of Operations

Full creation of an Office 365 account requires several steps, even when using the latest suggested solution from Microsoft. The latest suggested solution from Microsoft does not perform all of the steps, requiring manual intervention or an additional product.

The ideal way to create and manage Office 365 accounts is to use DirSync (legacy) or Azure AD Connect [68]. Both products will copy account details from the local AD DS to Azure AD. When an account is created, it is also created in Azure AD, and removed from Azure AD when an account is removed locally. Several attributes are synced to Azure AD, including phone, email, name, title, and department. By keeping AD DS up to date, Azure AD is kept up to date.

However, this only creates an Azure AD account, and does not apply any licensing or Exchange settings to the account. The assigning of licenses is left as "an exercise to the reader" by Microsoft. Suggestions include manually applying licenses, which does not work for a tenant of 37,000 accounts. The other suggestion is to batch license all accounts. This also does not work for a tenant of 37,000 accounts due to the amount of time it takes to process all 37,000 accounts every single time. It also assumes that a single unified license can be applied over all of the accounts, which is likely a false assumption. Current Microsoft licensing has a different license between employees and students for higher education.

To fully provision an Office 365 account, several steps are performed, all of which take advantage of the UAMS2 solution.

First an account is created in the local AD DS. The ordering is ensured by specifying `adauth` as a prerequisite for the Office 365 services. Once the AD DS account is created, then the provisioning request for Office 365 can be sent.

However, it is entirely possible that the DirSync process has not completed by the time this `create` request is processed. DirSync by default runs every three hours. This was too slow to be acceptable at NDSU, so it was dramatically reduced to around 15 minutes. However, failures in the DirSync process, timing oddities, and other issues may still mean that a DirSync has not run by the time the `create` is processed. Since the DirSync interval only affects account creation, the `create` requests are sent to a different queue. If the account does not exist, and a DirSync had not happened long enough after the `create` request, the message is NACK'd and is processed in the future. Since the queue is FIFO, all other messages behind that `create` message are no younger than the top of the queue, and thus would not exist either.

The main work queue for the driver is configured using the default Puppet type configuration. This means that it too sees all of the `create` requests. However, since it does not process `creates`, it simply ACK's each message without sending a response back to the dispatcher.

Once DirSync has happened and the account exists in Azure AD, it can be licensed. This licensing is performed using the Graph API [69], which is a web service using Open Data Protocol (ODATA) [70]. Current implementation uses Java to communicate with the Graph API. Once the account is licensed, a request can be sent back to the UAMS2 dispatcher indicating that the service has been created.

However, additional actions are required to completely configure an account given the current needs of the system. Those actions are performed on the Exchange mailbox that exists in Office 365. Those actions were not available to a public web service at time of need, and were only available via PowerShell cmdlets. Before the Office 365 driver sends back success, it sends messages with a different routing key, so that these three other processes will receive the request to update the accounts.

The three actions in no particular order are:

- Enable non-owner audit tracking on mailboxes and calendars.
- Enable email retention according to NDUS practices to be in compliance with retention requirements.
- Disable Clutter. Clutter is a feature that monitors which emails are most read and acted upon and filters out those that are not to the Clutter folder. This does not work well when admitted students get accounts, but do not start reading mail until their first semester. By that time, everything is marked and filtered into Clutter as they had not been reading email for months. Users can enable Clutter if they so choose after they start using their accounts. So it important that this change is only applied to new accounts.

All three of these actions are triggered on Exchange mailboxes. Normally the process to license an account quickly creates the mailbox. However, service issues in Office 365 have caused the mailbox to not be created for hours after the licensing request. Initially the code to handle these actions was configured to abort, email the developer, and ACK the message to try the next one after about 30 minutes. This was changed to try every 15 minutes and send an email each time a mailbox could not be found. If the emails persist for hours, the administrators can then check to see what the issue is, and see if workarounds need to be applied to get the mailbox to provision.

5.1.2. Chaining Services

Initial deployments of Office 365 included separate licenses for Office ProPlus, which granted those that were licensed the ability to install Microsoft Office on up to five systems. To get Office ProPlus, a user had to have Office 365. To get Office 365 a user had to already have AD DS.

Since Office ProPlus was only another license, it did not need any advanced handling around the DirSync timing issue. The `create` message would not be sent until Office 365 was active for the user, which meant that licensing had already succeeded. This means that the Office ProPlus licensing would not fail due to the account not being in Azure AD.

5.2. Inclusion in Web Applications

Many of the web applications in development at NDSU are designed to concentrate on one task. They are also designed to take advantage of Central Authentication Service (CAS) [71] by default, and build their entire security model around that system with the assistance of Apache Shiro [72].

Those web applications can also take advantage of IAM services. Since the applications are developed in Java, it is easy to have a thread dedicated in the background to handling the requests. Like all other drivers, it connects to RabbitMQ and processes the requests using all of the normal processing rules.

This keeps those operations near to the exact same code that is being used for everything else in the application. It does not require another code base with which to share the Object Relational Mapper (ORM) layer, and it does not require running a separate system or process. It also keeps the security model of the application concentrated on the use of CAS.

5.3. Compared to SSH

This system is replacing the legacy system that used SSH. Compared to it for the current needs of the institution, it is much better.

The original system controlled no more than eight or so services at any one time. Most of them were quick to operate on, and up until BPOS all were on Unix or Linux based systems. Also, up until BPOS, all of the services operated on local systems and not remote systems out of NDSU’s control. Since each service was custom in its dispatch and modification trigger code, the barrier to add a new service was moderately high. There was no graceful manner to handle unexpected failures on remote systems. At the time, the solution for local systems was to disable enrollments altogether when it was known that a system was going to be unavailable. If one of the local email systems was going to be down, all enrollments for all systems was disabled.

UAMS2 is currently responsible for just over 100 services, with more being added. The SSH based system with its sequential processing would not be able to keep up the number of requests that are sent through at times of high demand. The "fire and forget" method of this solution allows for requests to be sent out much quicker. While the LMS

deprovisioning process is slowly working through its queue, the UAMS2 dispatcher can continue to issue messages which allow the AD DS driver to rapidly work through its requests.

The standardization of messages makes it so that under most circumstances zero work has to be done on the dispatcher to add a service.

Given the generic nature of the technologies in use, most modern languages can be used. This allows the driver developer to choose the best language for their particular project. Drivers have been developed in Perl, PowerShell, Go, and Java. The slow to start Blackboard driver is still in Java, but it stays connected all the time and waits for a message to arrive in the queue. This makes `create`s and `modifie`s very quick to complete. `remove`s are still slow, but that is a result of the deletion process in Blackboard.

5.4. Compared to Connectors

Sun Identity Manager introduced their connector framework, which provided a new method for modifying services in the late 2000's. The new method was to connect the IdM to remote connectors. These connectors would either run in .NET or Java. The developer would place their custom .NET or Java code in the directory of the connector framework. This would allow local commands to be initiated by a remote system. If an organization was running Sun IdM on Solaris [73], they could use the .NET connector system to perform native tasks on a Windows Server.

Sun IdM has been phased out after the purchase of Sun by Oracle. However, the connector system was made available under at least one open source license. OpenICF is governed by ForgeRock and used in their OpenIDM product. OpenICF [74] is derived from Sun Identity Connector Framework (ICF). ConnID [75] is also derived from Sun ICF, and is managed through by Tisasa. It is the connector framework used to drive Apache Syncode and Evolveum midPoint, both of which are licensed under Apache License 2.0.

Remote hosts can run remote connectors in either .NET or Java, which covers most platforms. To operate, the connectors need to expose a port to allow communication to be sent through. To secure communication, TLS certificates need to be managed at each end point running a remote connector.

UAMS2 drivers connecting to RabbitMQ do not need to expose a port for a remote system to connect to. However, any host based firewall would need to allow the connection outbound. Additionally, as the driver is making an outbound connection, it does not need to manage TLS certificates. The only TLS certificate required is on the RabbitMQ broker.

The ability to scale and distribute work amongst load balanced connectors is dependent on the IAM solution driving the connectors.

5.5. Compared to Web Services

Web services act similar to connectors in many ways. There are roughly three ways that a system could be developed with remote web services.

First method is to run a remote web service to connect to. This is how many services UAMS2 interacts with work. However, instead of the driver connecting with the work queued up, the dispatcher would connect via the web service. It also requires running a web service on the remote system, and for that web service to handle authentication. While systems such as Google Apps have defined web services, NDSU Windows File service would require that it is developed from scratch. This would require exposing connections on the Windows Server system handling the changes. The dispatcher would need to handle retries when the web service is unavailable, and load balancing would have to be done via a HTTP load balancer. Finally, for slow actions, the system would need to be designed to avoid potential HTTP timeouts.

The second method would be a modification of the first. After the web service is dispatched, a callback URL could be included. Once the driver is done processing, it would send a request back to the callback URL, indicating the status. This would require that the driver have the logic and security configuration to perform that call back. Most web services run as a child of the HTTP system, where this asynchronous model would cause development complications.

Finally, the dispatcher could run the web service. The drivers would then connect to the dispatcher and request messages. This eliminates running HTTP servers on systems that generally should not be running HTTP servers. Balancing and issues with HTTP timeout remain. Using websockets would allow for a rapid response, similar to persistent RabbitMQ connections.

To directly connect to an existing web service, the dispatcher would need to be updated to be able to communicate with the web service. While many web services have standardize around SOAP 1.2 [76] or Representational State Transfer (REST) [77], the exact paths, request values, and message formats all differ. Some service providers provide implementations to interact with their web services. However, these implementations may not always match the languages directly available to the dispatcher, requiring a custom implementation. The UAMS2 solution allows the developer to choose a language with a provided API library, reducing the maintenance costs associated with that service.

While a combination of the web service options above would be useful in many situations, web services and the required HTTP connection management adds a layer of complexity beyond development in the proper language for the task and using RabbitMQ.

6. CONCLUSION

6.1. Issues

6.1.1. PowerShell

The .NET library can be used with PowerShell with some minor difficulty. .NET is case sensitive, as is C# and most of the programming languages .NET is used with. PowerShell is different in that it is not case sensitive. An if statement can be `if`, `IF`, `If`, or `iF`. The official .NET library for RabbitMQ uses `RabbitMQ.Client.ConnectionFactory` to create a connection in most use cases. This class exposes the following properties of concern [78]:

`public uri uri(w)`

`public string Uri(w)`

These two different properties vary in type and case. Unfortunately, being case insensitive, PowerShell cannot distinguish between the two. This requires that PowerShell reflection be used to setup the connection factory.

```
[Reflection.Assembly]::LoadFile("C:\Program Files (x86)\RabbitMQ\DotNetClient\
   bin\RabbitMq.Client.Dll")
[System.Reflection.Assembly]::LoadWithPartialName("System.Net.Security")
$factory = new-object RabbitMQ.Client.ConnectionFactory

$hostNameProp = [RabbitMQ.Client.ConnectionFactory].GetField("HostName")
$hostNameProp.SetValue($factory, $hostname)

$portProp = [RabbitMQ.Client.ConnectionFactory].GetField("Port")
$portProp.SetValue($factory, 5671)

$vhostProp = [RabbitMQ.Client.ConnectionFactory].GetProperty("VirtualHost")
$vhostProp.SetValue($factory, $port)

$usernameProp = [RabbitMQ.Client.ConnectionFactory].GetProperty("UserName")
$usernameProp.SetValue($factory, $config.configuration.username.value)

$passwordProp = [RabbitMQ.Client.ConnectionFactory].GetProperty("Password")
$passwordProp.SetValue($factory, $config.configuration.password.value)

$sslProp = [RabbitMQ.Client.ConnectionFactory].GetField("Ssl")
$sslOption = new-object RabbitMQ.Client.SslOption
$sslOption.enabled = $true
```

```
$sslProp.SetValue($factory, $sslOption)
$createConnectionMethod = [RabbitMQ.Client.ConnectionFactory].GetMethod("
   CreateConnection", [Type]::EmptyTypes)

# Create RabbitMQ connection
$connection = $createConnectionMethod.Invoke($factory, "instance,public", $null
   , $null, $null)
```

6.1.2. Java

While the Lyra library helps with reconnecting to RabbitMQ in the event of a failure, early versions of the library had an issue with one failure scenario. If the IP of the RabbitMQ broker became unavailable, as what happens if the system is rebooting, in Linux a `NoRouteToHostException` would be thrown. This was not handled as OS-X, the library's author's platform, does not throw that exception. It was further demonstrated that the exception was not thrown while the client was running on Microsoft Windows.

6.2. Future Work

6.2.1. Signing Requests

As implemented messages are sent to RabbitMQ over TLS, which protects the messages in transit, and the UAMS2 based system exists in its own vhost. Development and testing of new drivers are done on a separate vhost to ensure messages are not sent to the wrong driver. The messages are persisted to the disk in plain text.

This provides a level of security, but the security would be increased by signing and potentially encrypting the requests. Development work on user password changes in a different system in the ecosystem in use at NDSU has lead to the usage of JavaScript Object Signing and Encryption (JOSE), using JSON Web Signature (JWS) objects and JSON Web Encryption (JWE) objects [79].

The signed and/or encrypted message includes details of the cryptographic methods used to generate the message. Consumers of the message then know which methods are required to decrypt and/or validate the signature. The cryptographic methods include pre-shared keys and public key cryptography. Of interest to this process would be the public key based methods.

The UAMS2 dispatcher could sign the message with its private key. Drivers would then validate the signature with the dispatcher's public key. Additionally, the UAMS2 dispatcher could encrypt the message using the driver's public key. However, this would create a key distribution issue that would need to be addressed.

Initial deployment for password changing has shown good interoperability between the libraries in use for .NET/PowerShell, Go, and Java.

6.2.2. Reducing Timing

As deployed, the UAMS2 dispatcher runs via a cron job on a Linux host every five minutes. Messages are dispatched, and responses are checked. Due to the prerequisite system, the order of operations are:

1. Read responses
2. Send `create` messages
3. Send `modify` messages
4. Send `remove` messages
5. Read responses
6. Exit

The initial read responses is performed to catch messages that were sent while the process was not running. This may include `create` or `remove` responses that allow for new messages to be sent based on the prerequisites.

Additionally, most UAMS2 drivers run on a cron job or Windows scheduled task to check their work queues every five to ten minutes. These are usually staggered off of the dispatcher times. This can cause an update to take as much as 20 minutes to propagate through, more if prerequisites are also being processed.

This cron job / scheduled task method was chosen as it was the simplest to implement, particularly in Perl and PowerShell, but also Java. However, a Blackboard driver implementation, by Bryan Mesich, showed the power in maintaining a persistent connection. The Blackboard driver runs as a Linux daemon, and maintains an active connection to

RabbitMQ. For `creates` and `modifies`, the daemon was able to respond back with a success message faster than the dispatcher was able to update the database to indicate that the message was sent. This persistent connection model was then extended to Java web applications.

Changing all drivers to maintain persistent connections would dramatically reduce the response time for changes from minutes to sub-second in many cases.

To see the gains across the system, the UAMS2 dispatcher would need to dispatch messages at much smaller intervals. The best method of doing this would be to turn the dispatcher into a daemon as well.

While the existing dispatcher could run from cron at minute intervals, this does not entirely solve the problem. If a large number of requests are sent at the same time, the dispatcher could spend minutes sending all of the message out. This happens when 8,000 accounts are locked, and each account has at least seven services managed by UAMS2. After all requests are sent, the dispatcher would spend several minutes processing all of the responses.

A better solution would be to rewrite the dispatcher in a language that handles concurrent processing better. One possible solution would be to create a daemon written in Go. This daemon would have four goroutines [80]. Goroutines can be thought as extremely lightweight threads.

The goroutines would:

- Block on the response queue, handling driver responses as they appear
- Query the `enroll` table every N seconds, issuing new `create` requests as appropriate.
- Query the `to_modify` table every N seconds, issuing new `modify` requests as appropriate.
- Query the `bindings` table every N seconds, issuing new `remove` requests as appropriate.

SQL does not provide a mechanism to be notified on table changes. This requires that a query be issued every so often. A good value for N may be 15 seconds.

Very large changes, such as batch account locks after end of semesters, would still cause a delay in processing. However, while the `modifies` are being sent for the account locks, responses could be handled concurrently. If a `create` is requested during this process, that request would be pushed on to the driver's queue. That create would be handled faster than require that all modifies for all services be processed, followed up by all responses returned to that point.

REFERENCES

[1] (2016, Jan.) UH message broker. [Online]. Available: https://www.hawaii.edu/bwiki/display/UHIAM/UH+Message+Broker

[2] (2016) RabbitMQ - messaging that just works. [Online]. Available: https://www.rabbitmq.com

[3] (2016) Puppet - the shortest path to better software. [Online]. Available: https://puppet.com/

[4] OASIS, *SAML 2.0*, OASIS Std. saml-core-2.0-os, 2005. [Online]. Available: https://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf

[5] (2016) Incommon: Security, privacy and trust for the research and education community. [Online]. Available: https://www.incommon.org/

[6] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile," Internet Requests for Comments, Internet Engineeering Task Force, RFC 5280, May 2008. [Online]. Available: https://tools.ietf.org/html/rfc5280

[7] (2016) Active directory domain services. [Online]. Available: https://technet.microsoft.com/en-us/library/dd448614.aspx

[8] (2015, Dec.) True64 UNIX. [Online]. Available: https://en.wikipedia.org/wiki/Tru64_UNIX

[9] (2016) IBM AIX, UNIX, for IBM power systems. [Online]. Available: https://www-03.ibm.com/systems/power/software/aix/

[10] T. Ylonen and C. Lonvick, "The secure shell (ssh) protocol architecture," Internet Requests for Comments, Internet Engineeering Task Force, RFC 4251, January 2006. [Online]. Available: https://tools.ietf.org/html/rfc4251

[11] (2016) The linux kernel archives. [Online]. Available: https://www.kernel.org

[12] (2016) Oracle pl/sql. [Online]. Available:
http://www.oracle.com/technetwork/database/features/plsql/index.html

[13] (2016) Database 12c | oracle. [Online]. Available:
https://www.oracle.com/database/index.html

[14] ISO, *Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework)*, International Organization for Standardization Std. 9075-1:2011, 2011. [Online]. Available: http:
//www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=53681

[15] (2016) The perl programming language. [Online]. Available: https://www.perl.org

[16] (2016, May) Bash - GNU project - free software foundation. [Online]. Available:
https://www.gnu.org/software/bash/

[17] (2010) Chapter 56 identity connectors overview (sun identity manager 8.1 resources reference). [Online]. Available:
https://docs.oracle.com/cd/E19225-01/820-6551/giiwd/index.html

[18] (2016) MySQL. [Online]. Available: https://www.mysql.com/

[19] (2016) Blackboard. [Online]. Available: https://www.blackboard.com/

[20] (2016) Java software | oracle. [Online]. Available:
https://www.oracle.com/java/index.html

[21] (2014, Jul.) Windows server 2008 r2. [Online]. Available:
https://technet.microsoft.com/en-us/library/dd349801(v=ws.10).aspx

[22] (2016, May) Microsoft online services. [Online]. Available:
https://en.wikipedia.org/wiki/Microsoft_Online_Services

[23] (2016) Microsoft powershell. [Online]. Available:
https://msdn.microsoft.com/en-us/powershell/mt173057.aspx

[24] J. Sermersheim, “Lightweight directory access protocol (ldap): The protocol,” Internet Requests for Comments, Internet Engineeering Task Force, RFC 4511, June 2006. [Online]. Available: https://tools.ietf.org/html/rfc4511

[25] (2016) Secure enterprise email solutions for business | exchange. [Online]. Available: https://products.office.com/en-us/exchange/email

[26] (2016) Explore office 365 for business | office 365. [Online]. Available: https://products.office.com/en-us/business/explore-office-365-for-business

[27] (2016) Google for education: Save time and stay connected. [Online]. Available: https://www.google.com/edu/products/productivity-tools/

[28] (2016) Netiq edirectory | netiq. [Online]. Available: https://www.netiq.com/products/edirectory/

[29] *AMQP version 0-9-1*, AMQP Working Group Std. 0-9-1, 2008. [Online]. Available: http://www.amqp.org/specification/0-9-1/amqp-org-download

[30] (1999, Apr.) Mozilla public license 1.1. [Online]. Available: https://www.mozilla.org/en-US/MPL/1.1/

[31] (2016) Earlang programming language. [Online]. Available: https://www.erlang.org/

[32] (2011, Feb.) Rabbitmq » blog archive » introducing publisher confirms - messaging that just works. [Online]. Available: https://www.rabbitmq.com/blog/2011/02/10/introducing-publisher-confirms/

[33] S. Treadway. (2016) Github - streadway/amqp: Go client for AMQP 0.9.1. [Online]. Available: https://github.com/streadway/amqp

[34] D. Lambley. (2015, Mar.) Anyevent::rabbitmq - search.cpan.org. [Online]. Available: http://search.cpan.org/~dlambley/AnyEvent-RabbitMQ-1.19/lib/AnyEvent/RabbitMQ.pm

[35] J. Polo. (2014, Jul.) Download - uh identity and access managment - uh business wiki. [Online]. Available: https://www.hawaii.edu/bwiki/display/UHIAM/Download

[36] T. Dierks and E. Rescorla, “The transport layer security (tls) protocol version 1.2,” Internet Requests for Comments, Internet Engineeering Task Force, RFC 5246, August 2008. [Online]. Available: https://tools.ietf.org/html/rfc5246

[37] J. Halterman. (2016) Github - jhalterman/lyra: High availability rabbitmq client. [Online]. Available: https://github.com/jhalterman/lyra

[38] T. Bray, “The javascript object notation (json) data interchange format,” Internet Requests for Comments, Internet Engineeering Task Force, RFC 7159, March 2014. [Online]. Available: https://tools.ietf.org/html/rfc7159

[39] (2009, Oct.) YAML ain’t markup language (YAML) version 1.2. [Online]. Available: http://yaml.org/spec/1.2/spec.html

[40] Y. Shafranovich, “Common format and mime type for comma-separated values (csv) files,” Internet Requests for Comments, Internet Engineeering Task Force, RFC 4180, October 2005. [Online]. Available: https://tools.ietf.org/html/rfc4180

[41] *Extensible Markup Language (XML) 1.1 (Second Edition)*, World Wide Web Consortium Std. XML 1.1 Second Edition, 2006. [Online]. Available: https://www.w3.org/TR/xml11/

[42] U. Ogbuji. (2004, Mar.) Principals of xml design: When to use elements versus attributes. [Online]. Available: https://www.ibm.com/developerworks/library/x-eleatt/

[43] (2015, Aug.) Convertfrom-json. [Online]. Available: https://technet.microsoft.com/en-us/library/hh849898.aspx

[44] (2016) json - the go programming language. [Online]. Available: https://golang.org/pkg/encoding/json/

[45] (2004, Jan.) Apache license version 2.0. [Online]. Available: https://www.apache.org/licenses/LICENSE-2.0.txt

[46] (2016) Github - fasterxml/jackson: Main portal page for jackson project. [Online]. Available: https://github.com/FasterXML/jackson

[47] (2016) Github - google/gson: A java serialization/deserialization library that can conver java oobject into json and back. [Online]. Available: https://github.com/google/gson

[48] M. Hannyaharamitu. (2013, Oct.) JSON - search.cpan.org. [Online]. Available: http://search.cpan.org/~makamaka/JSON-2.90/lib/JSON.pm

[49] (2016) PHP: JSON. [Online]. Available: http://php.net/manual/en/book.json.php

[50] (2016) Red hat enterprise linux operating system. [Online]. Available: https://www.redhat.com/en/technologies/linux-platforms/enterprise-linux

[51] Github - pupppuppet/puppetlabs-rabbitmq. [Online]. Available: https://github.com/puppetlabs/puppetlabs-rabbitmq

[52] (2016) Ruby programming language. [Online]. Available: https://www.ruby-lang.org/en/

[53] (2016) Microsoft identity manageri 2016 | microsoft. [Online]. Available: https://www.microsoft.com/en-us/cloud-platform/microsoft-identity-manager

[54] (2016) Active directory - access & identity - idaas | microsoft azure. [Online]. Available: https://azure.microsoft.com/en-us/services/active-directory/

[55] (2015) Identity management product feature details. [Online]. Available: https://compare.evolveum.com/features-table.html

[56] (2016) midpoint identity & access managment system - evolveum. [Online]. Available: https://evolveum.com/midpoint/

[57] (2016, Jun.) Apache syncope. [Online]. Available: https://syncope.apache.org/

[58] (2016, Aug.) ASF committers by id. [Online]. Available: https://people.apache.org/committer-index.html

[59] (2016) Identity management from forgerock - openidm. [Online]. Available: https://www.forgerock.com/platform/identity-management/

[60] (2004, Dec.) Common development and distribution license (CDDL-1.0)| open source initiative. [Online]. Available: https://opensource.org/licenses/CDDL-1.0

[61] (2016) Elixir. [Online]. Available: http://elixir-lang.org/

[62] A. Videla and J. J. W. Williams, *RabbitMQ in Action: Distributed Messaging for Everyone*, 1st ed. Shelter Island, NY: Manning Publications, 2012.

[63] (2016) Nagios - the industry standard in it infrastructure monitoring. [Online]. Available: https://www.nagios.org/

[64] D. Bode, *Puppet Types and Providers*. Sebastopol, CA: O'Reilly Media, 2012.

[65] (2016) The world's leading research & insights platform | qualtrics. [Online]. Available: https://www.qualtrics.com/

[66] (2016) Fargo bikeshare. [Online]. Available: http://greatridesbikeshare.org/

[67] (2016) Faculty activity reporting made easy, effective and secure | digital measures. [Online]. Available: http://www.digitalmeasures.com/

[68] A. Kjellman. (2016, Jun.) Azure ad connect: Upgrade from dirsync | microsoft azure. [Online]. Available: https://azure.microsoft.com/en-us/documentation/articles/active-directory-aadconnect-dirsync-upgrade-get-started/

[69] (2016) Microsoft graph - home. [Online]. Available: https://graph.microsoft.io/en-us/

[70] *OData Version 4.0*, OASIS Std. OData Version 4.0. Part 1: Protocol Plus Errata 03, Feb. 2014. [Online]. Available: http://docs.oasis-open.org/odata/odata/v4.0/os/part1-protocol/odata-v4.0-os-part1-protocol.pdf

[71] (2016) Single sign-on for the web. [Online]. Available: https://apereo.github.io/cas/

[72] (2016) Apache Shiro | java security framework. [Online]. Available: https://shiro.apache.org

[73] (2016) Solaris 11 | security. speed. simplicity. | oracle. [Online]. Available: https://www.oracle.com/solaris/solaris11/index.html

[74] (2016) OpenICF - forgerock community. [Online]. Available: https://forgerock.org/openicf/

[75] (2015, Oct.) ConnID. [Online]. Available: http://connid.tirasa.net/

[76] *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*, World Wide Web Consortium Std. SOAP 1.2, 2007. [Online]. Available: https://www.w3.org/TR/soap12-part1/

[77] R. T. Fielding, “REST: architectural styles and the design of network-based software architectures,” Doctoral dissertation, University of California, Irvine, 2000. [Online]. Available: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[78] (2016) public class connectionfactory. [Online]. Available: https://rabbitmq.com/releases/rabbitmq-dotnet-client/v3.6.3/rabbitmq-dotnet-client-3.6.3-client-htmldoc/html/type-RabbitMQ.Client.ConnectionFactory.html

[79] M. Jones, J. Bradley, and N. Sakimura, “Json web token (jwt),” Internet Requests for Comments, Internet Engineeering Task Force, RFC 7519, May 2015. [Online]. Available: https://tools.ietf.org/html/rfc7519

[80] (2016) Effective go - the go programming language. [Online]. Available: https://golang.org/doc/effective_go.html#goroutines